```
#include "mex.h"

void mxHisto(const mxArray* source, mxArray* dest, int bin);
void mxCreateCumulativeHistogram(unsigned int *pDest);

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    const int bitDepth = 256; // number of max intensity values
    int binHeight;
    int binWidth;
    int bin;
    int dims[2];

    // check paramters: nlhs, nrhs
    if (nlhs > 2 || nrhs < 2 || nrhs > 2)
        mexErrMsgTxt("Usage: histo = mxHisto(Image, binSize)");

    // check input: uint8
    if (!mxIsUint8(prhs[0]))
        mexErrMsgTxt("mxHisto: Image must be uint8");

    binHeight = mxGetM(prhs[1]);
    binWidth = mxGetN(prhs[1]);

    // check bin: double, complex, height and width
    if (!mxIsDouble(prhs[1]) || mxIsComplex(prhs[1])
            || !(binHeight == 1 && binWidth == 1))
        mexErrMsgTxt("binSize must be non-complex scalar uint8");

    bin = mxGetPr(prhs[1])[0];
    dims[0] = binHeight * binWidth; // height
    dims[1] = bitDepth / bin; // width

    // check bin: 1 <= bin <= 256
    if (!(1 < bin < bitDepth))
        mexErrMsgTxt("Bin size must be between 1 and 256");


    // input / output

    // create output array
    // output
    plhs[0] = mxCreateNumericArray(2, dims, mxUINT32_CLASS, mxREAL);

    // create histogram: mxHisto(...);
    mxHisto(prhs[0], plhs[0], bin);
}

void mxHisto(const mxArray* source, mxArray* dest, int bin)
{
    const int bitDepth = 256; // number of max intensity values
    //get dimensions of image
    const int height = mxGetM(source);
    const int width = mxGetN(source);
    const int dimension = height * width;
```

```
    int newIntensity;

    unsigned char *pSource = (unsigned char*) mxGetData(source);
    unsigned int *pDest = (unsigned int*) mxGetData(dest);

    for (int i = 0; i < dimension; i++) {
        newIntensity = pSource[i] / bin; // integer operations only!
        pDest[newIntensity]++;
    }

    mxCreateCumulativeHistogram(pDest);
}

void mxCreateCumulativeHistogram(unsigned int *pDest)
{
    for (int i = 1; i < 256; i++) {
        pDest[i] += pDest[i-1];
    }
}
```

```
% ------------------------------------------------------------------------
%   Uebung 3, Di Martino, Stefano, 286021,
%   Semester 7, stemarti@htwg-konstanz.de, 15. April 2014
%
%   Uebung 3, Willhelm, Andreas, 286297,
%   Semester, anwilhel@htwg-konstanz.de, 15. April 2014
%
%   Uebung 3, Kocher, Theresa,
%
% ------------------------------------------------------------------------
Aufgabe 2:
- Aufgabe 2.1: Was ist ein Filter?
Durch einen Filter kann man ein Bild bearbeiten, verfremden oder verändern.
Bei einem Filter wird ein Pixel aus den umliegenden Nachbarschaftspixel neu
berechnet. Dadurch kann man z.B. ein Bild glätten.

- Aufgabe 2.2: Was ist der Unterschied zwischen Punktoperationen und
Filteroperationen?
Bei der Punktoperation wird auf einen genau Pixel eine Operation angewandt.
Dabei werden umliegende Pixel nicht verändert oder miteinbezogen.
Dadurch kann man z.B. die Helligkeit eines Bildes anheben.
Bei der Filteroperation wird ein Pixelwert mit Hilfe der umliegende
Nachbarschaftspixel neu berechnet. Dadurch kann man z.B. ein Bild glätten
oder auch eine Histogrammanpassung vornehmen.
```

```
/* ----------------------------------------------------------------------
%   Uebung 3, Di Martino, Stefano, 286021,
%   Semester 7, stemarti@htwg-konstanz.de, 15. April 2014
%
%   Uebung 3, Willhelm, Andreas, 286297,
%   Semester, anwilhel@htwg-konstanz.de, 15. April 2014
%
%   Uebung 3, Kocher, Theresa,
%
% ----------------------------------------------------------------------
*/

#include <math.h>
#include "mex.h"

static const int intensities = 256;

// returns the cumul. distribution function for histogram h
static void createCumulativeDistributionFunction ( unsigned int *h, double *P) {
    double hValue = 0.0, sum = 0.0; // cumulate histogram values

    for (int i = 0; i < intensities; i++) {
        sum += h[i];
    }

    for (int i = 0; i < intensities; i++) {
        hValue += h[i]; //cumulative
        P[i] = (double) hValue / sum; //normalize
    }
}

// returns the mapping function F () to be applied to image I A
static void adaptHisto (unsigned int *hReference, unsigned int *hOriginal, unsigned↙
int *hAdapt) {
    double POriginal[intensities];
    double PReference[intensities];

    createCumulativeDistributionFunction(hReference, PReference); // create CDF table
    createCumulativeDistributionFunction(hOriginal, POriginal); // create CDF table

    int newIntensity;
    // compute mapping function f hs ():
    for (int intensity = 0; intensity < intensities; intensity++) {
        newIntensity = intensities - 1;

        do {
            hAdapt[intensity] = newIntensity;
            newIntensity--;
        } while (newIntensity >= 0 && POriginal[intensity] <= PReference↙
[newIntensity]);
    }
}
```

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    int mrows1, ncols1, mrows0, ncols0;
    int dims[2];

    // check: nlhs ,nrhs
    mrows1 = mxGetM(prhs[1]);
    ncols1 = mxGetN(prhs[1]);
    mrows0 = mxGetM(prhs[0]);
    ncols0 = mxGetN(prhs[0]);

    if (!(mrows0 == 1 && ncols0 == 256)) {
        mexErrMsgTxt("Look-up table size must be [1][256]");
    }
    if (!(mrows1 == 1 && ncols1 == 256)) {
        mexErrMsgTxt("Look-up table size must be [1][256]");
    }
    if (mrows1 != mrows0 || ncols1 != ncols0) {
        mexErrMsgTxt("The given array must have the same size");
    }

    // check paramters: nlhs, nrhs
    if (nlhs > 1 || nrhs != 2)
        mexErrMsgTxt("Usage: mxHistoAdapt(refHisto, adaptHisto);");

    // input / output
    dims[0] = mrows0;
    dims[1] = ncols0;

    // create output array
    plhs[0] = mxCreateNumericArray(2, dims, mxUINT32_CLASS, mxREAL);

    unsigned int *phRef = (unsigned int*) mxGetData(prhs[0]);
    unsigned int *phOrig = (unsigned int*) mxGetData(prhs[1]);
    unsigned int *pLUT = (unsigned int*) mxGetData(plhs[0]);

    adaptHisto(phRef, phOrig, pLUT);
}
```